

# Partie 4 : Les principaux types

## Chaînes de caractères

### 1. Syntaxe

La chaîne de caractères est absolument indispensable à tout programme informatique : elle permet au programme de communiquer avec ses utilisateurs en leur donnant des informations. Cependant, il s'agit d'un objet assez complexe, puisqu'une chaîne doit pouvoir être très malléable et permettre, par exemple, de contenir certaines parties variables.

Pour écrire un littéral chaîne de caractères, il est possible d'utiliser indifféremment des guillemets droits simples ou doubles :

```
chaine = "chaîne"
chaine = "chaîne"
```

Les deux objets ainsi créés sont exactement identiques.

Il faut aussi savoir qu'en Python, on laisse la possibilité d'écrire des chaînes sur plusieurs lignes, en procédant ainsi :

```
"""
Ceci est une chaîne sur plusieurs lignes.
Ceci est une nouvelle ligne.
"""
```



Chose intéressante, si une telle chaîne est non assignée à une variable et déclarée tout en haut du module, elle sera la documentation du module aussi appelée docstring. Idem pour une fonction ou une classe.

```
def fonction():
    """
        Ceci est la documentation de la fonction
    """
    help(fonction)
```

### 2. Formatage d'une chaîne

Nous avons déjà vu que la méthode permettant d'afficher dans la sortie standard peut prendre 1 à n paramètres :

```
print(chaine, nombre, autre_chaine)
```

Cependant, ceci n'est pas la seule manière de réaliser des affichages complexes, et sans forcément passer par la sortie standard.

Pour formater des chaînes, Python s'est inspiré de C :

```
"Tu t'inspires de %s ?" % "C" # Renvoie Tu t'inspires de C ?
```

Pour ceci, on utilise l'opérateur modulo, mais cet opérateur ne prend que deux opérandes : la chaîne à formater à gauche et les variables à injecter à droite. S'il y a plusieurs variables à injecter, il faut donc utiliser un n-uplet :

```
"Veux-tu la %(color)s %(obj)s ?" % {"pilule": "bleue"}  
# Renvoie 'Veux-tu la pilule bleue ?'
```

Cette méthode est très facile à utiliser, mais rencontre un problème essentiel : si l'on doit afficher plusieurs fois la même variable, il faut la mettre plusieurs fois dans le n-uplet, et si l'on doit traduire notre chaîne, il faut le faire d'une manière qui respecte l'ordre des variables à injecter, sans quoi les trous sont remplis dans le mauvais ordre.

En effet, en général, dans ce genre de situation, l'opérande de gauche, c'est-à-dire la chaîne de caractères à formater, est un littéral dans le code, qui peut être traduit à l'aide d'un outil comme gettext. L'opérande de droite est pour sa part constitué par des variables qui, elles aussi, peuvent être traduites de leur côté.

Pour faciliter les choses, il faut alors utiliser un dictionnaire. Voici un exemple en français :

```
"Veux-tu la %(obj)s %(color)s ?" % {"obj": "pilule",  
                                      "color": "bleue"}  
# Renvoie 'Veux-tu la pilule bleue ?'
```

Et en anglais :

```
>>> "Do you want the %(color)s %(obj)s ?" % {"obj": "pill",  
                                              "color": "blue"}  
# Renvoie 'Do you want the blue pill ?'
```

Le formatage de chaîne à l'aide de l'opérateur modulo est universellement utilisé dans tous les modules Python. Cependant, Python pousse à l'utilisation d'une nouvelle méthode, inspirée cette fois-ci de C++ :

```
>>> "Veux-tu la {} {}".format("pilule", "bleue")  
# Renvoie 'Veux-tu la pilule bleue ?'
```

Cette méthode permet de gérer la position des variables :

```
>>> "Veux-tu la {0} {1} ?".format("pilule", "bleue")  
# Renvoie 'Veux-tu la pilule bleue ?'  
>>> "Do you want the {1} {0} ?".format("pill", "bleue")  
# Renvoie 'Do you want the blue pill ?'
```

Et il est aussi possible de nommer ces arguments :

```
>>> "Veux-tu la {obj} {color} ?".format(obj="pilule", color="bleue")  
# Renvoie 'Veux-tu la pilule bleue ?'  
>>> "Do you want the {color} {obj} ?".format(obj="pill", color="blue")  
# Renvoie 'Do you want the blue pill ?'
```

Nous allons donc privilégier, autant que possible, l'utilisation de cette méthode pour nos nouveaux développements.

### 3. Notion de casse

Les caractères ont une notion de casse, qui s'applique aux lettres, accentuées ou non. Voici comment transformer une chaîne de caractères pour la mettre en minuscules :

```
chaine_minuscules = chaine.lower()
```

Si vous ne deviez retenir qu'une seule chose de ce chapitre, ce serait ceci : **la zone mémoire d'une chaîne de caractères n'est jamais modifiée.**

Dans cet exemple précédent, la méthode `lower` va travailler sur la chaîne, mais pour vous en renvoyer une nouvelle, à un autre emplacement mémoire et avec la transformation que vous avez demandée. L'objet d'origine n'est pas modifié en mémoire, il ne le sera jamais.



Une chaîne de caractères est un objet **non mutable**. C'est une des notions les plus importantes à comprendre lorsque l'on débute en Python.

Par conséquent, après la transformation, votre variable pointe vers une zone mémoire différente. C'est pour cela que ce qui suit ne suffit pas :

```
chaine.lower()
```

Pour vous en convaincre :

```
chaine = "Test"  
chaine_minuscule = chaine.lower()  
print(chaine)  
print(chaine_minuscule)
```

Enfin, si on écrit ceci :

```
chaine = "Test"  
chaine = chaine.lower()
```

Alors il n'y a plus de pointeur vers la zone mémoire contenant "Test", puisque la variable `chaine` pointe maintenant vers la zone mémoire contenant "`test`". Cette zone mémoire est déclarée orpheline et sera nettoyée lors du prochain passage du ramasse-miettes, le dispositif de Python pour recycler la mémoire qui n'est plus utilisée. Sachez que ce dispositif existe, qu'il gère toutes les problématiques de gestion de mémoire à votre place et qu'il le fait de façon optimisée.

Il existe aussi une méthode pour mettre la chaîne en majuscules :

```
chaine_majuscule = chaine.upper()
```



On pourra également citer les méthodes `capitalize` ou `title` ainsi que `swapcase`, que vous êtes invité à découvrir en analysant le fichier Guide/21\_Chaines/21\_01\_Introduction.py.

### 4. Notion de longueur

La longueur d'une chaîne de caractères s'obtient ainsi :

```
longueur = len(chaine)
```

Cette façon de faire est typique de la programmation impérative. Pour un langage purement objet, on se serait attendu à faire quelque chose comme `chaine.len()`, mais cela n'est pas le cas en Python.

Pourquoi ? D'une part parce que la doctrine de Python précise : « Il doit y avoir une, et de préférence une seule, façon évidente de faire les choses ». D'autre part pour des raisons de consistance du langage : si `len` était une méthode, rien n'empêcherait qu'elle porte un nom différent d'une classe à l'autre.

Il faut cependant savoir que la fonction `len` s'appliquera automatiquement à tout objet mesurable, c'est-à-dire ayant une longueur ; nous verrons pourquoi ainsi que les mécanismes sous-jacents dans le chapitre Modèle objet de la partie Les fondamentaux du langage.

La notion de longueur de caractères est une notion de haut niveau. Pour ceux qui ont l'habitude des langages de bas niveau, en Python on compte le nombre de caractères, et non le nombre d'octets utilisés pour les représenter, et sans compter le caractère de fin de chaîne.

```
len("Flèche: -->") # Renvoie 9
```

Eh oui, la table Unicode est gigantesque et l'on y trouve des symboles, pas uniquement des lettres.

### 5. Appartenance

Python a une méthode très simple pour savoir si une chaîne (nommée `morceau`) est contenue dans une autre (nommée `chaine`) :

```
morceau in chaine
```

L'utilisation du mot-clé `in` et non d'un quelconque opérateur ou méthode de classe est également un choix structurant du langage : il faut privilégier la lisibilité.

En lisant le code, on comprend tout de suite « le morceau est dans la chaîne ? » et la réponse sera `True` ou `False`.

## 6. Notion d'occurrence

Compter le nombre d'occurrences d'un caractère dans une chaîne se fait en utilisant une méthode :

```
chaine = 'abcdaefabcefab'
chaine.count('a') # Renvoie 4
```

Mais notez que l'on peut compter aussi des morceaux de chaînes :

```
chaine = 'abcdaefabcefab'
chaine.count('ab') # Renvoie 3
```

Voici un autre exemple :

```
chaine = 'abcdaefabcefab'
chaine.count('abc') # Renvoie 2
```

Il est aussi possible de trouver l'indice de la position de la première occurrence d'un caractère :

```
position = phrase.index("a")
```

 La première position d'un caractère dans une chaîne est toujours 0 et la dernière  $n - 1$ ,  $n$  étant la longueur de la chaîne.

Pour trouver la position suivante, il faut utiliser la même fonction, mais en lui demandant de commencer la recherche à l'indice suivant :

```
position2 = phrase.index("a", position + 1)
```

S'il n'y a plus de positions à trouver, la méthode renverra la valeur  $-1$ . Comme précédemment, on n'a pas seulement la possibilité de rechercher des caractères, mais également des chaînes de caractères.

Voici un algorithme permettant d'afficher toutes les positions :

```
def positions(chaine, morceau):
    position = -1
    for i in range(chaine.count(morceau)):
        position = chaine.index("a", position + 1)
        print("Position de la n°{} :{}".format(i + 1, position))
```

On note que l'on réutilise toujours le même appel de la fonction `index`, mais que l'on doit commencer par la position 0. Or, comme on utilise `position + 1`, on déclare la position initiale à  $-1$  pour pouvoir commencer au premier caractère.

Pour trouver la position suivante, il faut utiliser la même fonction, mais en lui demandant de commencer la recherche à l'indice suivant :

```
position2 = phrase.index("a", position + 1)
```

S'il n'y a plus de positions à trouver, la méthode renverra la valeur  $-1$ . Comme précédemment, on n'a pas seulement la possibilité de rechercher des caractères, mais également des chaînes de caractères.

Voici un algorithme permettant d'afficher toutes les positions :

```
def positions(chaine, morceau):
    position = -1
    for i in range(chaine.count(morceau)):
        position = chaine.index("a", position + 1)
        print("Position de la n°{} :{}".format(i + 1, position))
```

On note que l'on réutilise toujours le même appel de la fonction `index`, mais que l'on doit commencer par la position 0. Or, comme on utilise `position + 1`, on déclare la position initiale à  $-1$  pour pouvoir commencer au premier caractère.

Sachez que pour rechercher un morceau dans une chaîne, il faut faire exactement pareil, mais avec la méthode `find`.

## 7. Remplacement

Python dispose d'une méthode permettant de remplacer des caractères :

```
chaine.replace("a", "A")
# Renvoie 'AbcdAefAbcefAb'
```

Ceci fonctionne également pour des chaînes de caractères :

```
>>> chaine.replace("ab", "AB")
# Renvoie 'ABcdacfABcefAB'
```

Et rien ne nous oblige à ce que la chaîne de remplacement ait la même longueur que celle de recherche :

```
>>> chaine.replace("abc", "[--0--]")
# Renvoie '[--0--]daef[--0--]efab'
```

## 8. Notion de caractère

Pour Python, il n'y a pas besoin d'avoir un type pour représenter un caractère. Un caractère est simplement une chaîne de caractères de longueur 1.

Pour Python, les chaînes de caractères sont encodées en utilisant l'Unicode. Autrement dit, chaque caractère est positionné dans une table et dispose d'une position dans cette table, nommée ordinal.

Ainsi, la lettre 'A' a pour ordinal 65 et l'ordinal 97 correspond à la lettre 'a'. Tous les caractères accentués ainsi que les signes de ponctuation ont également un ordinal, mais celui-ci peut aller au-delà de 255. En effet, en Unicode, un caractère peut être encodé en 1, 2, 3 ou 4 octets. Fort heureusement, il n'est nul besoin de se préoccuper de ces problématiques bas niveau lorsque l'on développe en Python, mais sachez que vous pourrez mettre dans vos chaînes des caractères étrangers, comme le ñ espagnol (<https://fr.wikipedia.org/wiki/%C3%91>) ou l'eszett allemand. Voici une liste de caractères spécifiques :

```
[chr(x) for x in range(191, 564)]
```

On pourra trouver des e cédille majuscule ou d'autres particularités des langues indo-européennes, mais également l'ensemble des lettres de l'arabe ou de l'hébreu, les idéogrammes indiens, chinois ou japonais (<https://en.wikipedia.org/wiki/Katakana>, voir le bas de la fiche, rubrique Unicode) ou encore ceux des langues mortes comme le nabatéen (ordinaux 67712 à 67759) ou le phénicien (67840 à 67871).

Notons que l'on peut accéder à un caractère précis d'une chaîne à l'aide de l'opérateur crochet :

```
mot[0]
```

Enfin, il existe aussi des symboles dans la table Unicode, lesquels sont d'ailleurs utilisés en CSS pour faire de la mise en forme.



Le fichier Guide/21\_Chaines/21\_02\_caractères.py vous permettra de faire vos propres essais.

## 9. Typologie des caractères

De la même manière que les nombres ont le module `math` qui contient les fonctions essentielles pour eux, les chaînes de caractères ont le module `unicodedata`:

```
import unicodedata
unicodedata.category('a') # Renvoie 'Ll'
unicodedata.category('A') # Renvoie 'Lu'
unicodedata.category('é') # Renvoie 'Ll'
unicodedata.category('É') # Renvoie 'Lu'
unicodedata.category('ç') # Renvoie 'Ll'
unicodedata.category('â') # Renvoie 'Ll'
unicodedata.category(chr(0x10880)) # Renvoie 'Cn'
unicodedata.category('>') # Renvoie 'Sm'
```

On peut donc savoir assez facilement si un caractère est une lettre minuscule ou majuscule en testant sa catégorie. Pour tester les autres catégories, il faut cependant avoir une bonne connaissance de l'Unicode ([https://en.wikipedia.org/wiki/Unicode\\_character\\_property#General\\_Category](https://en.wikipedia.org/wiki/Unicode_character_property#General_Category)).

C'est pourquoi, même aujourd'hui alors que l'Unicode est partout, on a plutôt tendance à utiliser le module `string`.

Ce dernier contient quelques chaînes contenant tous les caractères d'un type particulier :

```
import string
string.ascii_letters
# Renvoie 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
string.ascii_lowercase
# Renvoie 'abcdefghijklmnopqrstuvwxyz'
string.ascii_uppercase
# Renvoie 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Ainsi que les chiffres (selon la base) :

```
string.digits
# Renvoie '0123456789'
string.hexdigits
# Renvoie '0123456789abcdefABCDEF'
string.octdigits
# Renvoie '01234567'
```

La ponctuation et les espaces :

```
string.punctuation
# Renvoie '!"#$%&\'()*+,-./;:<>?@[\\\]^`{|}~'
string.whitespace
# Renvoie '\t\n\r\x0b\x0c'
```

Ainsi que l'ensemble des précédents :

```
string.printable
# Renvoie '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!#$%&\'()*+,-./;:<>?@[\\\]^`{|}~\t\n\r\x0b\x0c'
```

Ce module est idéal pour la langue anglaise, mais pour travailler avec la langue française, il faudra souvent l'adapter pour y faire apparaître nos caractères supplémentaires, lesquels seront spécifiques et pourront être différents de ceux d'autres langues pourtant proches, comme l'espagnol ou l'italien.

```
If lettre in string.digits :
    print("la lettre {} est un chiffre.".format(lettre))
```

## 10. Séquencer une chaîne de caractères

Voici comment découper une chaîne de caractères :

```
mots = phrase.split()
```

Le découpage se fait par rapport à l'ensemble des caractères blancs (`string.whitespace`). On obtient ainsi une liste de chaînes de caractères que l'on peut recomposer en les cimentant à l'aide d'une chaîne de caractères qui va faire office de glu :

```
"<glu>".join(mots)
```

La chaîne glu peut tout aussi bien être une chaîne de caractères vide.

Découper caractère par caractère une chaîne de caractères peut se faire très simplement :

```
liste_caracteres = list(chaine_de_caracteres)
```

On obtient ainsi une liste, la même chose que ce que l'on avait obtenu en utilisant la méthode `split` et que l'on va détailler à présent.



Vous trouverez un fichier pour vous aider à tester cela : Guide/21\_Chaines/21\_03\_séquençage.py.

# Listes

## 1. Syntaxe

La liste est l'objet conteneur par excellence. Au contraire de la chaîne de caractères, elle est modifiable et elle est même faite pour être modifiée.

Elle peut contenir tout type d'objet, y compris des objets de types différents.

Commençons par créer une liste de caractères :

```
liste = list("Python is awesome")
```

Ce qui équivaut à déclarer la liste ainsi :

```
liste = ['P', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 'a',
         'w', 'e', 's', 'o', 'm', 'e']
```

Les délimiteurs de la liste sont les crochets et chaque élément est séparé de son voisin par une virgule.

## 2. Indices

Chaque élément de cette liste dispose d'un indice (comme la chaîne de caractères) et ces indices sont extrêmement importants, ils peuvent être utilisés de beaucoup de manières.

Voici pour commencer comment récupérer un élément :

```
liste[4] = o
```

Mais aussi comment utiliser un indice négatif pour partir de la fin :

```
liste[-3] = o
```

On peut réaliser ce que l'on appelle une extraction de sous-liste, qui consiste à créer une copie partielle d'une liste, par exemple du début, avec les six premiers éléments :

```
liste[:6] = ['P', 'y', 't', 'h', 'o', 'n']
```

Les deux-points servent à séparer l'indice de début et celui de fin.

On peut également faire une extraction des septième et huitième caractères :

```
liste[7:9] = ['i', 's']
```

Là encore, il s'agit d'un élément important : en Python, les indices permettant de délimiter une zone vont toujours du premier inclus au dernier exclu.

Pour aller jusqu'à la fin, on peut laisser le second argument vide :

```
liste[10:] = ['a', 'w', 'e', 's', 'o', 'm', 'e']
```

Enfin, on peut également utiliser un pas, en positionnant un troisième argument :

```
liste[2::5] = ['t', 'i', 'e']
```

Et on peut faire tout ceci indifféremment en utilisant des indices et un pas qui peuvent être positifs ou négatifs.

```
liste[-3::-6] = ['o', 's', 't']
```

Attention cependant : on part du premier élément et on se déplace dans le sens du pas (vers la droite si le pas est positif). Si l'indice d'arrivée est dans la direction opposée, la sous-liste obtenue sera vide.

L'indice permet également d'assigner une nouvelle valeur à cet emplacement de la liste :

```
liste[11] = "b"
```

On peut également remplacer plusieurs éléments d'un coup (attention, la longueur des éléments doit être égale des deux côtés) :

```
liste[13:15] = "fg"
```

Et on peut supprimer un élément de la liste :

```
del liste[15]
```

Attention, la longueur de la liste sera réduite de 1.

On peut également supprimer plusieurs éléments d'un coup :

```
del liste[:7]
```

La longueur de la liste sera réduite d'autant.

### 3. Valeurs

Tout comme pour la chaîne de caractères, on peut rechercher une occurrence à l'aide de la méthode `index` et leur nombre à l'aide de la méthode `count`.

On peut aussi demander à supprimer un élément particulier de la liste (ne sachant pas son indice) :

```
liste.remove("y")
```

Comme on peut le constater, on n'écrit pas :

```
liste = liste.remove("y")
```

En effet, la liste est un objet **mutable**, c'est-à-dire que son espace mémoire est modifiable et que l'on peut travailler en son sein. Par conséquent, ses méthodes ne renvoient rien, ou plutôt renvoient `None`, l'élément vide de Python. Ainsi, ce dernier exemple est une erreur de développement, puisque l'on perd le pointeur vers notre liste qui n'est donc plus atteignable.



Attention à cette notion fondamentale **mutable/non mutable**. Tester les différentes méthodes des chaînes (non mutables) et des listes (mutables) permet de visualiser de quelle manière il faut les traiter.

Voir tous les supprimer :

```
while " " in liste:  
    liste.remove(" ")
```

Enfin, on peut utiliser la liste comme une pile en supprimant une valeur à la fin et en la retournant :

```
liste.pop()
```

Et en rajoutant une valeur à la fin :

```
liste.append("h")
```

On peut également rajouter des valeurs à n'importe quel endroit, il suffit de préciser l'indice d'insertion et la valeur à insérer :

```
liste.insert(2, "d")
```

On peut enfin ajouter une autre liste, à la fin :

```
liste.extend(["i", "j"])
```



Pour prendre en main la liste, nous vous proposons d'exécuter le fichier `Guide/22_Listes/22_01_Introduction.py` et de chercher à comprendre ce qu'il s'y passe, étape par étape.



Exercice : à la fin du fichier, vous verrez deux fonctions incomplètes. Vous devrez utiliser les méthodes que vous souhaitez (en ne travaillant que sur les indices ou sur les valeurs) pour arriver au résultat attendu, à partir de la liste de départ.

```
def exercice1():  
    liste = ["P", "t"]  
    # TODO  
    assert "".join(liste) == "Python"  
  
def exercice2():  
    liste = [1, 4, 2, 5, 4, 3, 4, 7, 5, 8, 9]  
    # TODO  
    assert liste == list(range(1, 6, 2))
```

## 4. Hasard

Savoir manipuler des listes est primordial. Un élément important à connaître est qu'en Python, il n'est absolument pas nécessaire de se référer aux indices pour manipuler les données d'un tableau. On peut le faire mais on en a rarement besoin.

Ainsi, dans les langages bas niveau, lorsque l'on veut choisir un élément au hasard dans une liste, on calcule la longueur de la liste, on choisit un nombre (toujours entre 0 et 1), on multiplie ces deux données et on garde la partie entière pour avoir l'indice de l'objet aléatoirement sélectionné.

En Python, rien de tout cela.

Pour vous le prouver, jouons à un jeu de cartes :

```
cartes = [chr(x) for x in range(0x1f0a1, 0x1f0af)]
```

soit, grâce à la magie de l'Unicode :



Et pour choisir une carte :

```
from random import choice  
choice(cartes)
```

On peut aussi en sélectionner un certain nombre, par exemple 5 :

```
sample(cartes, 5)
```

Et enfin, on peut tout simplement les mélanger :

```
shuffle(cartes)
```

Tout comme pour le tri, que vous verrez très prochainement, la liste elle-même est modifiée. Écrire `cartes = shuffle(cartes)` serait une erreur (testez-la pour en être sûr !).



Vous trouverez le résultat associé à tous ces nombreux exemples en exécutant le fichier Guide/22\_Listes/22\_02\_Hasard.py.

## 5. Techniques d'itération

Comme nous l'avons dit précédemment, savoir manipuler des listes est primordial et il n'est absolument pas nécessaire de se référer aux indices pour manipuler les données d'un tableau.

Pour faire simple, prenons une liste simple et courte :

```
liste = list("abc")
```

Voici comment itérer dans cette liste :

```
for lettre in liste:  
    print(lettre)
```

Pas de notion d'indice ici, la variable `lettre` va contenir directement l'élément de la liste en question. Ceci signifie qu'à l'intérieur du bloc itératif, on n'a aucune idée de la position de l'élément dans la liste, et il faut bien préciser que, la plupart du temps, on n'a pas du tout besoin de le savoir.

Mais si vraiment on en a besoin :

```
for indice, lettre in enumerate(liste):  
    print("indice {}, lettre {}".format(indice, lettre))
```

Le générateur `enumerate` va nous générer cette position, puisqu'à chaque boucle il va renvoyer deux valeurs : dans l'ordre, l'indice et le caractère.

Notons aussi la syntaxe particulière de cette ligne, car ces deux valeurs vont aller respectivement dans les variables `indice` et `lettre`.

Il est temps maintenant de passer au tableau (ici, le terme "tableau" désigne une liste de listes). Voici ce qu'il ne faut pas faire :

```
tableau = [liste, liste]
```

Pourquoi ?

```
tableau[0][0] = "X"  
print("tableau = {}".format(tableau))  
# Renvoie : tableau = [['X', 'b', 'c'], ['X', 'b', 'c']]  
print("liste = {}".format(liste))  
# Renvoie : ['X', 'b', 'c']
```

Qu'a-t-il bien pu se passer ?

C'est assez simple : la liste est un élément mutable. Elle est donc transformable.

Lorsque nous écrivons :

```
a = [1, 2, 3]
b = a
```

les identificateurs **a** et **b** pointent vers la même zone mémoire. Donc, si nous modifions l'une des variables, nous modifions l'autre, automatiquement.

C'est ce qu'il s'est passé là. La même case mémoire est pointée par **liste[0]**, **tableau[0][0]** et **tableau[1][0]**.

Repronons :

```
tableau = [liste[:], [c.upper() for c in liste]]
```

Ici, nous avons utilisé une extraction de sous-liste du début à la fin, soit une copie de la liste, puis une compréhension de liste pour avoir une copie de la liste, mais avec des lettres majuscules.

Voici maintenant comment faire une itération :

```
for ligne in tableau:
    for case in ligne:
        print(case)
```

Cependant, en Python, on n'aime pas trop faire des boucles imbriquées, donc on préférera utiliser un générateur tel que celui-ci :

```
from itertools import chain
for case in chain.from_iterable(tableau):
    print(case)
```

Ce générateur va itérer par lui-même successivement sur toutes les lignes, mais d'une manière plus performante.

Cependant, si vraiment nous avons besoin des indices, il est possible de faire ceci :

```
for i, ligne in enumerate(tableau):
    for j, case in enumerate(ligne):
        print("tableau[{}][{}] = {}".format(i, j, case))
```

Mais il faut savoir que dans la quasi-totalité des cas, il est possible de s'en passer en utilisant quelques astuces.

Itérer sur les lignes est sympathique, mais on a parfois besoin d'itérer sur les colonnes. Et on ne veut pas mettre en place de solutions trop complexes.

Heureusement pour nous, il est possible de faire la transposée d'un tableau :

```
transpose = zip(*tableau)
```

Il suffit alors d'itérer sur les lignes du tableau pour itérer sur les colonnes du tableau :

```
for j, colonne in enumerate(zip(*tableau)):
    for i, case in enumerate(colonne):
        print("tableau[{}][{}] = {}".format(i, j, case))
```

Les fonctions `zip` et `enumerate` sont toutes les deux des générateurs. Python va donc travailler sur chaque élément en mémoire sans avoir besoin de calculer et stocker la transposée. Ceci lui permet de garder des performances correctes sur ce genre d'opération.

Pour terminer, sachez que l'on a souvent affaire à des données qui représentent des cases positionnées sur un tableau, mais que la manière de les stocker ne suit pas un ordre, pour des raisons de performance.

Sachez que la plupart du temps, vous pourrez vous recréer artificiellement ces lignes et ces colonnes en un tour de main :

```
from itertools import product

lignes = ["A", "B", "C"]
colonnes = [1, 2, 3]

for ligne, colonne in product(lignes, colonnes):
    print("Case {}{}".format(ligne, colonne))
```

Nous réutiliserons cette astuce.

Si l'on n'a qu'une seule ligne, on peut soit utiliser la méthode précédente :

```
from itertools import product
for ligne, colonne in product(["Z"], colonnes):
    print("Case {}{}".format(ligne, colonne))
```

Soit virtualiser une liste ne contenant que l'élément souhaité, mais le nombre voulu de fois :

```
from itertools import repeat
for ligne, colonne in zip(repeat("Z"), colonnes):
    print("Case {}{}".format(ligne, colonne))
```

Enfin, on peut vouloir répéter une séquence, mais un nombre indéterminé de fois pour en avoir autant que nécessaire :

```
from itertools import cycle
for numero, lettre in zip(range(10), cycle("ABC")):
    print("Case {}{}".format(lettre, numero))
```



Vous trouverez le résultat associé à tous ces nombreux exemples en exécutant le fichier Guide/22\_Listes/22\_03\_Iterations.py.

## 6. Tri

Une des problématiques essentielles en programmation consiste à trier des données. En Python, ceci se fait assez simplement :

```
liste = [0, 3, 7, 8, 2, 4, 1, 6, 5, 9]
liste.sort()
```

Il est important de voir que la méthode `sort` va trier la liste en place, mais ne va rien renvoyer, au contraire de toutes les méthodes vues pour la chaîne de caractères : c'est parce que la liste est mutable.

On peut donc afficher la liste pour constater qu'elle est bien triée.

Pour les nombres, il n'y a pas de soucis, mais lorsque le tri a une signification plus subtile, c'est une autre paire de manches. Voici une liste de chaînes de caractères :

```
mots = "Ah La phrase à trier a été déclarée".split()
```

Voici le résultat :

```
mots.sort() # Transforme la liste en :
['Ah', 'La', 'a', 'déclarée', 'phrase', 'trier', 'à', 'été']
```

En réalité, le tri se fait sur l'ordinal des caractères. Par conséquent, on commence par toutes les majuscules, puis les minuscules et enfin les accents.

Pour s'en sortir sans avoir à écrire des algorithmes complexes, il faut savoir utiliser les subtilités permises par Python.

En effet, on peut passer en paramètre de la méthode `sort` une clé qui sera appliquée à chaque élément de la liste et la comparaison se fera non plus sur les éléments eux-mêmes, mais sur les éléments transformés à l'aide de cette clé.

Or, il est assez facile de mettre les caractères en minuscules, à l'aide de leur méthode `lower`. Mais cette méthode appartient en fait à leur classe, qui est `str`. Elle est accessible via `str.lower` et utilisable comme une fonction.

En d'autres termes, utiliser "`A`".`lower`() est la même chose que `str.lower("A")`.

Et pour Python, tout est objet, les chaînes comme les nombres ou les fonctions ou encore les classes. Nous pouvons donc créer une variable pointant sur une fonction :

```
ma_fonction = str.lower
```

Vous noterez qu'il n'y a pas de parenthèses et que, par conséquent, il ne s'agit pas d'un appel de fonction. On dit que `ma_fonction` est simplement la même chose que `str.lower`. Et on peut l'utiliser :

```
ma_fonction("A")
```

Tout cela pour dire qu'il est possible de passer `str.lower` en tant que clé de comparaison :

```
mot.sor(key=str.lower) # Renvoie :  
['a', 'Ah', 'déclarée', 'La', 'phrase', 'trier', 'à', 'été']
```

Reste à traiter le problème des accents.

Pour cela, il faut indiquer pour tous les caractères accentués leur correspondance avec le caractère non accentué. Ceci se fait en utilisant un dictionnaire de traduction, mais heureusement pour nous, il est simple à déclarer :

```
translation = str.maketrans(  
    "àââééèéîîôôùùýýçç_~",  
    "aaaeeeiiiioouuuuyyc ",  
    "#~.?:!")
```

Tous les caractères de la première ligne seront remplacés par ceux qui leur sont immédiatement en dessous et tous les caractères de la dernière ligne seront purement et simplement supprimés.

Il faut donc maintenant écrire la fonction de transformation :

```
def transformation(x):  
    return x.lower().translate(translation)
```

Et passer cette fonction en paramètre :

```
mot.sor(key=transformation) # Renvoie  
['a', 'à', 'Ah', 'déclarée', 'été', 'La', 'phrase', 'trier']
```

 Vous trouverez ces exemples dans le fichier Guide/22\_Listes/22\_04\_Tri.py.

Pour information, le dictionnaire de traduction ressemble à cela :

```
{63: None, 46: None, 95: 32, 224: 97, 33: None, 226: 97, 35: None,  
228: 97, 231: 99, 232: 101, 233: 101, 234: 101, 235: 101, 44:  
None, 45: 32, 238: 105, 239: 105, 59: None, 244: 111, 246: 111,  
375: 121, 249: 117, 58: None, 251: 117, 252: 117, 126: None, 255:  
121}
```

Les clés sont les ordinaux des caractères à remplacer et les valeurs les ordinaux des caractères de remplacement, ou `None` s'il faut les supprimer.

Il est donc temps de voir ce qu'est un dictionnaire.

# Dictionnaires

## 1. Présentation des dictionnaires

Un dictionnaire est un conteneur qui associe une clé à une valeur. Il est un type de données essentiel lorsque l'on souhaite accéder très rapidement à une valeur. Pour l'illustrer, imaginons un carnet d'adresses à l'ancienne : pour chaque nom, on associe un numéro de téléphone.

```
carnet = {  
    "Clément": "0601020304",  
    "Chloé": "0102030405",  
    "Matthieu": "0205080502",  
}
```

On a donc trois entrées dans ce dictionnaire. Le plus important à se rappeler est que la seule chose qui compte est l'association entre la clé et la valeur.

Ainsi, si l'on souhaite obtenir le numéro de téléphone de Chloé, on peut procéder ainsi :

```
carnet["Chloé"]
```

L'accès à l'élément est extrêmement rapide, beaucoup plus que d'aller chercher un élément dans une liste ordonnée, triée par ordre des prénoms, par exemple.

On peut ajouter un nouveau numéro ainsi :

```
carnet["Sébastien"] = "0408060204"
```

Si la clé existait, alors sa valeur est mise à jour ; sinon, un nouvel enregistrement est créé.

Là encore, cette opération est très rapide. Et il en est ainsi parce qu'il n'y a pas d'ordre dans un dictionnaire et on n'a d'ailleurs pas besoin de maintenir un ordre à tout instant.

## 2. Parcourir un dictionnaire

Voici comment parcourir un dictionnaire :

```
for nom, telephone in carnet.items():  
    print("Le numéro de {} est {}".format(nom, telephone))
```

Et si vraiment on doit absolument parcourir le dictionnaire dans l'ordre, on devra alors itérer sur les clés, mais en ayant pris soin de les trier auparavant :

```
for nom in sorted(carnet.keys()):  
    print("Le numéro de {} est {}".format(nom, carnet["nom"]))
```

La valeur est obtenue en allant la chercher dans le dictionnaire à chaque itération.

On peut tester la présence d'une clé facilement :

```
"Cassiopée" in carnet
```

Et on accède à une entrée du dictionnaire, même si on n'est pas certain qu'elle y soit :

```
carnet.get("Cassiopée")
```

Et demander à ce qu'une valeur par défaut soit renvoyée si la clé n'existe pas :

```
carnet.get("Cassiopée", "0987654321")
```

Les avantages de cette structure de données sont assez édifiants et elle est très complémentaire avec la liste. Si on utilise uniquement ces deux conteneurs, on peut représenter à peu près tout ce que l'on veut.

### 3. Exemple

Voici un exemple inspiré du Black Jack :

```
cartes = {  
    chr(0x1f0a1): 11,  
    chr(0x1f0a2): 2,  
    chr(0x1f0a3): 3,  
    chr(0x1f0a4): 4,  
    chr(0x1f0a5): 5,  
    chr(0x1f0a6): 6,  
    chr(0x1f0a7): 7,  
    chr(0x1f0a8): 8,  
    chr(0x1f0a9): 9,  
    chr(0x1f0aa): 10,  
    chr(0x1f0ab): 10,  
    chr(0x1f0ad): 10,  
    chr(0x1f0ae): 10,  
}
```

Ici, le dictionnaire sert à obtenir la valeur de chaque carte.

Il faut créer à partir de ce dictionnaire une liste de cartes, qu'on utilisera pour pouvoir choisir une carte :

```
liste_cartes = list(cartes)
```

On peut maintenant faire choisir au joueur deux cartes, l'une après l'autre :

```
from random import choice, sample  
  
carte = choice(liste_cartes)  
score = cartes[carte]  
  
carte = choice(liste_cartes)  
score += cartes[carte]
```

À chaque étape, on rajoute le score de la carte piochée, que l'on obtient très facilement.

Puis faire choisir deux cartes au hasard à la banque :

```
main_banque = sample(liste_cartes, 2)  
score_banque = sum(cartes[carte] for carte in main_banque)
```

Ici, on utilise une expression génératrice (similaire à l'expression à l'intérieur d'une compréhension de liste) pour additionner les valeurs des deux cartes.

Et voici un exemple d'exécution :

```
Vous avez choisi: A ♠ 10 ♠ >>> votre score est de 21  
La banque a: 3 ♠ 10 ♠ >> son score est de 13
```



Vous pourrez tester ce programme et l'adapter, il se trouve dans le fichier Guide/23\_Dictionnaire/23\_01\_Introduction.py.